Optimal Memory Management for Time Warp Parallel Simulation

Yi-Bing Lin
Bellcore
Morristown, NJ 07962-1910

Bruno R. Preiss
Department of Electrical
and Computer Engineering
University of Waterloo
Waterloo, Ontario
Canada, N2L 3G1

Abstract

Recently there has been a great deal of interest in performance evaluation of parallel simulation. Most work is devoted to the time complexity and assumes that the amount of memory available for parallel simulation is unlimited. This paper studies the space complexity of parallel simulation. Our goal is to design an efficient memory management protocol which guarantees that the memory consumption of parallel simulation is of the same order as sequential simulation. (Such an algorithm is referred to as *optimal*.) We first derive the relationships among the space complexities of sequential simulation, Chandy-Misra simulation, and Time Warp simulation. We show that Chandy-Misra may consume more storage than sequential simulation, or vice versa. Then we show that Time Warp never consumes less memory than sequential simulation. Then we describe *cancelback*, an optimal Time Warp memory management protocol proposed by Jefferson. Although cancelback is considered as a complete solution for the storage management problem in Time Warp, some efficiency issues in implementing this algorithm must be considered. In this paper, we propose an optimal algorithm called *artificial rollback*. We show that this algorithm is easy to implement and analyze. An implementation of artificial rollback is given, which is integrated with processor scheduling to adjust the memory consumption rate based on the amount of free storage available in the system.

1 Introduction

A discrete event simulation consists of a series of events, with times when they occur. Execution of an event can give rise to any number of events with later timestamps. The simulation is straightforward to implement if there is a centralized system with one event queue — just execute the earliest not-yet-executed event next.

Since simulation is time-consuming, it is natural to attempt to use multiple processors to speed up the simulation process. In *parallel discrete event simulation* (or *parallel simulation*), the simulated system is partitioned into a set of sub-systems that are simulated by a set of processes that communicate by sending and receiving timestamped messages. The scheduling of an event for a sub-system at time t is simulated by sending a message with timestamp t to the corresponding process. The global event list and global clock of a sequential simulation do not exist in the parallel counterpart. Each process has its own input message queue and local clock. To correctly simulate a sub-system, the corresponding process must execute arriving messages in their timestamp order, as opposed to their real-time arrival order. To satisfy this causality constraint, a synchronization mechanism is required. Two of the most common synchronization protocols for parallel simulation are the *Chandy-Misra* protocol [2] and the *Time Warp* protocol [7]. (Different approaches for parallel simulation are discussed elsewhere [5, 10, 13, 16, 22, 26, 27].) An introduction to the Chandy-Misra protocol and the Time Warp protocol can be found in [5].

Recently there has been a great deal of interest in performance evaluation of parallel simulation. Most work [3, 4, 13, 14, 15, 18, 19, 20, 21, 23, 24, 29] is devoted to the time complexity and assumes that the amount of memory available for parallel simulation is unlimited. This paper studies the space complexity of parallel simulation. We derive the relationships of the space complexities among sequential simulation, Chandy-Misra simulation, and Time Warp simulation. We show that Chandy-Misra may consume more storage than sequential simulation, or vice versa. Then we show that Time Warp always consumes more memory than sequential simulation. The derivations indicate that parallel simulation may consume much more memory than sequential simulation. Thus, it is important to design an efficient memory management protocol which guarantees that the memory consumption of parallel simulation is of the same order as sequential simulation. (Such an algorithm is referred to as an *optimal* memory management algorithm; a formal definition will be given in Definition 1.) Previous work [6, 7, 8] has been devoted to reducing the space complexity of Time Warp simulation. The first optimal memory management protocol (called *cancelback*) was proposed by Jefferson [8]. Although cancelback is considered as a complete solution for the storage management problem in Time Warp, some efficiency issues in implementing this algorithm must be considered. In this paper, we propose an optimal algorithm called *artificial rollback*. We show that this algorithm is easy to implement and analyze. An implementation of artificial rollback is given, which is integrated with processor scheduling to adjust the memory consumption rate based

on the amount of free storage available in the system.

All notations used in this paper are listed in Appendix A.

2 Memory Usage in Parallel Simulation

This section gives the relationships among the space complexities of sequential simulation, Chandy-Misra simulation, and Time Warp simulation. We show that (i) Chandy-Misra may consume more storage than sequential simulation, or vice versa, and (ii) Time Warp always consumes more memory than sequential simulation. Consider a K-process simulation[1]. Let $ts(e)$ be the timestamp[2] of an event e. Without loss of generality, we assume that all events executed within a process have different timestamps. Let Ψ be the set of events executed in a sequential simulation. Then Ψ is the set of events executed in Chandy-Misra and is the set of committed events in the corresponding Time Warp simulation. Let $x_i(\tau)$ be the state of process p_i after it has executed all events with timestamp no later than τ. Suppose that the scheduling of an event e is due to the execution of another event e_0. Then the *send time* of e (denoted by $ts'(e)$) is defined as the timestamp of e_0. In other words, $ts'(e) = ts(e_0)$. Since the execution of an event always schedules events with later timestamps, we have $ts'(e) = ts(e_0) < ts(e)$. Let $E(\tau)$ be the set of events in the event queue of the sequential simulation when all events with timestamp earlier than τ have been executed. Then $E(\tau) = \{e \in \Psi | ts'(e) < \tau, ts(e) \geq \tau\}$. For a given timestamp value τ, let $\tau_i^- = ts(e)$ be the least timestamp of the event $e \in \Psi_i$ such that $\tau_i^- < \tau$, and for all $e' \in \Psi_i$, $ts(e') < \tau \Rightarrow ts(e) \leq \tau_i^-$. After p_i has executed all events with timestamps earlier than τ, its process state is $x_i(\tau_i^-)$. Thus, the system state of the sequential simulation at τ (or the *sequential snapshot at* τ before the event with timestamp τ is executed) is

$$\mathcal{M}_s(\tau) = \left[\bigcup_{1 \leq i \leq K} x_i(\tau_i^-), E(\tau) \right] \tag{1}$$

Let an *item* be a message or a copy of process state. Let $M_s(\tau) = |\mathcal{M}_s(\tau)|$ be the amount of storage required to store all items in $\mathcal{M}_s(\tau)$. Then the storage consumed by the sequential simulation is $M_s = \max_{\forall \tau} |\mathcal{M}_s(\tau)|$.

[1] We consider object-oriented simulation. Thus, the concept of "process" exists in sequential simulation.

[2] In the remainder of this paper, the term *time* means real time, and the term *timestamp* means simulation time.

Definition 1 Suppose that a sequential simulation consumes M_s units of memory. The corresponding parallel simulation consumes *constant bounded* memory if and only if its space complexity is $O(M_s)$. A memory management algorithm is *optimal* if it ensures that every parallel simulation consumes constant bounded memory.

2.1 Chandy-Misra

Consider a Chandy-Misra simulation at time t. Let $x_i(t)$ be the state of process p_i at time t. We assume that $|x_i(t_1)| = |x_i(t_2)| = |x_i|$ for all t_1, t_2 (i.e., the size of a process does not change). Note that if all events with timestamps no later than τ have been executed by p_i at time t, then $x_i(t) = x_i(\tau)$. Let $I_i(t)$ be the set of events scheduled for p_i at time t; i.e., the events already in p_i's input queue, and the events that have been sent from other processes, but not yet received by p_i. Let $O_i(t)$ be the output queue of process p_i at time t. Then the *snapshot of Chandy-Misra at time t* is

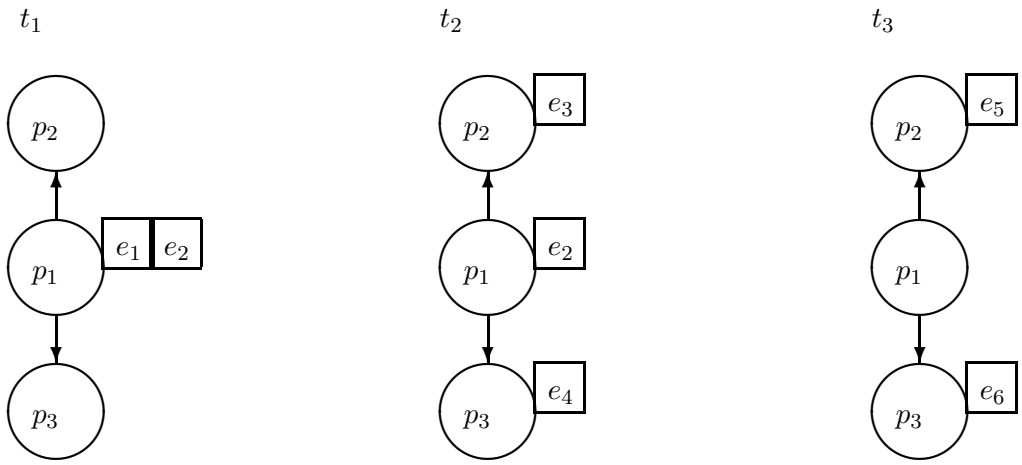
$$\mathcal{M}_{cm}(t) = \bigcup_{1 \le i \le K} [x_i(t), I_i(t), O_i(t)]$$

and the storage consumed by the Chandy-Misra simulation is $M_{cm} = \max_{\forall t} |\mathcal{M}_{cm}(t)|$.

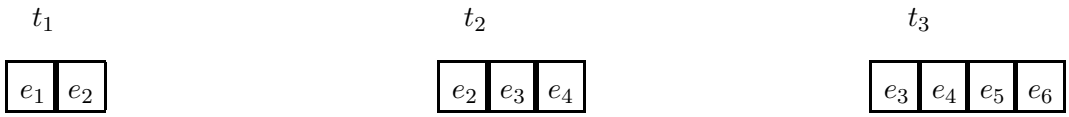
Lemma 1 There exists a simulation such that $M_{cm} < M_s$.

Proof: Consider a 3-process simulation where a *source* process p_1 sends messages (i.e., schedules events) to two *sink* processes p_2 and p_3 (cf. Figure 1 (a)). Let $ts(e_i) = i$. Initially, two events e_1 and e_2 are scheduled for p_1. The execution of e_1 results in the scheduling of e_3 and e_4 respectively. The execution of e_2 results in the scheduling of an event e_5 to process p_2 and an event e_6 to process p_3. The executions of e_3, e_4, e_5 and e_6 do not schedule any new events. Consider a Chandy-Misra simulation. If the memory used for process states is the same for both Chandy-Misra and sequential simulations, then we only need to consider the memory used for event queues. Suppose that (i) the lookahead of p_1 is 2, which is known in advance, (ii) the event execution time is a constant, and (iii) message sending delay is 0. Then the maximal amount of memory consumed by the input queues in the Chandy-Misra simulation is 3 (cf. Figure 1 (a)). On the other hand, the maximal amount of memory used in the corresponding sequential simulation is 4 (cf. Figure 1 (b)). ∎

Now we show that Chandy-Misra may consume non-constant bounded memory. Lin and Lazowska [17], and Jefferson [8] show that there exist Chandy-Misra simulations that have space



(a) Storage consumed in a Chandy-Misra simulation.



(b) Storage consumed in a sequential simulation.

Figure 1: The case when Chandy-Misra consumes less storage than the sequential simulation.

complexities of $O(kM_s)$ for arbitrary k. The following lemma generalizes the previous results by showing that Chandy-Misra may have exponential space complexity compared with the sequential simulation.

Lemma 2 For every positive integer k, there exists a simulation such that the space complexity of Chandy-Misra is $O(M_s^k)$.

Proof: Consider the simulation of a $(k+1)$-level tree structure process network. Every subroot (and the root) has k branches. Thus the root of the tree is a source process, and the leaves of the tree are sink processes, and each subroot has one input channel and k output channels. The simulation behaves as follows:

- One event is scheduled to the root at the beginning. The execution of the event results in the scheduling of $2k$ events, two for each of its children.
- Every subroot p_i will receive two messages (events) from its parent. The execution of the event with earlier timestamp results in the scheduling of $2k$ events, two for each of its children. The execution of the other event results in the scheduling of k events for p_i itself.
- Every leaf will receive two messages (events) from its parent. The execution of the first event results in the scheduling of k events to the leaf itself. The execution of the second event does not schedule any new event.
- For every subtree of a subroot (or the root) p_i, all events created and/or executed in that subtree have timestamps earlier than the events scheduled to p_i itself.
- For every two subtrees rooted at p_i, all events created and/or executed at one subtree have timestamps earlier than that at the other subtree.

Suppose that the execution of an event takes one unit of time, and the message sending delay is ignored. Then the space complexity of the sequential simulation is $O(k^2)$, and the space complexity for the Chandy-Misra simulation (with any size of input buffer) is $O(k^{k+1})$ or $O(M_s^{k'})$ for $k' = k-1$. ∎

2.2 Time Warp

Consider Time Warp. Let $X_i(t)$ be the set of states in the state queue of process p_i at time t. Let $I_i(t)$ be the set of events scheduled for p_i at time t; i.e., the events already in p_i's input queue (including the input messages that have already been processed but not yet discarded by fossil collection; the definition for fossil collection will be given later)[3] and the events that have been sent from other processes, but not yet received by p_i. Let $O_i(t)$ be the output queue (i.e., the collection of negative messages) of process p_i at time t. (Note that the definition of the output queue in Time Warp is different from that in Chandy-Misra.) Then the *snapshot of Time Warp at time t* is

$$\mathcal{M}_{tw}(t) = \bigcup_{1 \le i \le K} [X_i(t), I_i(t), O_i(t)]$$

and the storage consumed by the Time Warp simulation is $M_{tw} = \max_{\forall t} |\mathcal{M}_{tw}(t)|$.

Let $ck_i(t)$ be the local clock of process p_i at time t; that is, $ck_i(t)$ is the timestamp of the event executed by p_i at time t. Let an *unprocessed* message be a message in transit or a message in the the input queue of a process, but not yet executed. The *global virtual time* (GVT) is defined as follows.

Definition 2 GVT at time t (denoted as GVT(t)) is the minimum of (i) the values of all local clocks at time t, and (ii) the timestamps[4] of all unprocessed messages.

It is difficult to obtain $GVT(t)$ in practice (especially in a distributed environment). In general, only lower bounds for $GVT(t)$ can be computed [12, 28]. We assume that the true $GVT(t)$ can be obtained. The results in this paper generalize for the case where only a lower bound for $GVT(t)$ is available. It is apparent that at any real time there exists a global virtual time GVT such that all executed messages with timestamps earlier than GVT cannot be rolled back. Since GVT is no larger than the timestamp of every unprocessed message in the system, we have the following theorem [7].

Theorem 1 (i) At time t, any event with timestamp earlier than GVT(t) cannot be rolled back, and may be irrevocably committed with safety. (ii) GVT is a non-decreasing function of time which

[3] Note that in Chandy-Misra simulation, $I_i(t)$ does not include any message which is already executed.

[4] In the original definition of GVT given by Jefferson [7], the send times of unprocessed messages are considered in computing GVT, instead of their timestamps. This paper follows the definition given in Fujimoto [5], and Lin and Lazowska [12].

guarantees global progress of the Time Warp simulation.

The following two corollaries are direct consequences of Theorem 1. Let $ts(x)$ be the timestamp of a process state x ($ts(x) = ts(e)$ if the process state of p_i is x after it executes e).

Corollary 1 Let $\tau \leq GVT(t)$. After time t, the following items in a process p_i are obsolete and can be deleted:

- The messages with timestamps earlier than τ in the input queue.
- The copies of process state with timestamps no later than τ except for the one with the largest timestamp no later than τ. From Theorem 1, this process state is $x_i(\tau)$ if the process has completed the execution of an event with timestamp τ. Otherwise, the state is $x_i(\tau_i^-)$.
- The messages with send times earlier than τ in the output queue.[5]

A *fossil collection* is performed up to simulation time τ at time t, if all obsolete items in Corollary 1 are discarded, and the storage for these items is reclaimed at time t.

Corollary 2 Consider a Time Warp simulation. Suppose that fossil collection is performed up to simulation time $\tau_{FC}(t) \leq GVT(t)$ at time t. Consider a timestamp τ such that $\tau_{FC}(t) \leq \tau \leq GVT(t)$. Let $I_{i,\tau}(t) = \{e \in I_i(t) | ts'(e) < \tau\}$, and let Ψ_i be the set of events executed by p_i in the sequential simulation. (That is, $\bigcup_{1 \leq i \leq K} \Psi_i = \Psi$.) Then

$$I_{i,\tau}(t) = \{e \in \Psi_i | ts'(e) < \tau, ts(e) \geq \tau_{FC}(t)\}$$

Proof: From Corollary 1, for every $e \in I_{i,\tau}(t)$, $ts(e) \geq \tau_{FC}(t)$. From Theorem 1 and because Time Warp has the same semantics as the sequential simulation, we have

$$I_{i,\tau}(t) = \{e \in \Psi_i | ts'(e) < \tau, ts(e) \geq \tau_{FC}(t)\} \ \blacksquare$$

Lemma 3 For all simulations, $M_{tw} \geq M_s$.

[5] In fact, output messages with send times equal to τ can also be discarded. For our purpose, we assume that these messages are not discarded. This assumption will slightly increase the amount of memory consumed by memory management protocols such as cancelback and artificial rollback.

Proof: Consider a Time Warp simulation. Suppose that fossil collection is performed up to simulation time $\tau_{FC}(t) \leq GVT(t)$ at time t. Consider a timestamp τ such that $\tau_{FC}(t) \leq \tau \leq GVT(t)$. From Corollary 2,

$$I_{i,\tau}(t) = \{e \in \Psi_i | ts'(e) < \tau, ts(e) \geq \tau_{FC}(t)\}$$

Since $E(\tau) = \{e \in \Psi | ts'(e) < \tau, ts(e) \geq \tau\}$, we have

$$E(\tau) \subseteq \bigcup_{1 \leq i \leq K} I_{i,\tau}(t) \subseteq \bigcup_{1 \leq i \leq K} I_i(t)$$

This implies that for all t, and for any τ such that $\tau_{FC}(t) \leq \tau \leq GVT(t)$, we have

$$M_{tw}(t) \geq M_s(\tau) \tag{2}$$

(Note that Time Warp consumes more storage for process states than sequential simulation does.) From the definition of GVT, we have

$$\text{for all } t, \quad GVT(t) \in \{ts(e) | e \in \Psi\} \tag{3}$$

Based on (2) and (3), we have

$$\text{for all } \tau \text{ there exists } t \text{ such that } M_{tw}(t) \geq M_s(\tau)$$

That is, $M_{tw} \geq M_s$. ∎

The intuition behind Lemma 3 is that Time Warp always contains a sequential snapshot in order to support rollback. Thus, more storage must be used in Time Warp. Lemma 3 holds for Time Warp with memory management protocols such as the *optimal checkpoint interval* approach [11, 25], fossil collection, *cancelback* protocols [6, 8], or the *artificial rollback* protocol to be described later.

The following lemma holds for Time Warp if fossil collection is the only means for memory management (the proof is omitted).

Lemma 4 For all positive integer functions $f(x) > x$, there exists a Time Warp simulation with fossil collection such that its space complexity is $O(f(M_s))$.

Although Chandy-Misra tends to consume less memory than Time Warp, it is not difficult to find examples where Time Warp is more economic in memory usage. Furthermore, to our knowledge,

there is no optimal memory management protocol for Chandy-Misra simulation. On the other hand, such protocols exist for Time Warp. Section 3 describes Gafni's protocol and the cancelback protocol. Gafni's protocol is not optimal. However, it does reduce the amount of memory required in a Time Warp simulation. The cancelback protocol is optimal in a shared memory architecture. Section 4 proposes an optimal protocol (for a shared memory architecture) called *artificial rollback*. The significance of the cancelback and artificial rollback protocols is that, while a Chandy-Misra simulation may fail due to non-constant bounded memory usage in a shared memory computer system with $O(M_s)$ storage, a Time Warp simulation will survive.

3 Cancelback Protocols

This section describes Gafni's protocol [6] and the cancelback protocol [8]. We first define the term "memory exhaustion".

Definition 3 Consider a Time Warp simulation with memory management algorithm Y. Let $F(t)$ be the amount of free storage available at time t. The system exhausts the storage if and only if $F(t) < \delta(t)$ at some t before the simulation completes, where $\delta(t)$ is the minimal amount of storage required to make the simulation progress (i.e., to advance GVT) at time t, and algorithm Y cannot produce more storage such that $F(t') \geq \delta(t)$ for all $t' \geq t$.

From Definition 3, a Time Warp simulation does not necessarily exhaust storage when $F(t) < \delta(t)$. With the memory management algorithms to be described in this paper, free storage could be reclaimed, and the simulation may continue.

The basic idea is to cancel "possibly-correct computation", if necessary, to obtain more free memory. In Gafni's protocol (cf. Figure 2), if fossil collection cannot reclaim enough free memory to satisfy the request of a process p, then some stored item of p has to be removed to make more room. The cancelled item will be re-produced later. If e is an input message (i.e., a positive message), then it is removed from p's input queue, and sent in the reverse direction back to its sender q's output queue. Message e is annihilated with its antimessage present in the output queue (most likely), and possibly causes a rollback at q. If e is an output message (i.e., a negative message), then it is removed from the output queue, and transmitted forward to annihilate its antimessage

```
while p needs more storage do
/* let Γ_p be the set of items in p with send times larger than GVT */
    if Γ_p ≠ ∅ then
        select the item e with the largest send time from Γ_p;
        case e.type of
            INPUT_MESSAGE:
                return e to its sender; discard e;
            OUTPUT_MESSAGE:
                transmit e; discard e;
            STATE:
                discard e;
        end case;
    else
        call block-mode routine;
    end if
end while
```

Figure 2: Gafni's Protocol.

(the positive copy). If e is a state, then it is deleted, and p rolls back to the state preceding e.

Gafni's algorithm assumes that every process has its own memory space. In the following example, we show that even if all processes share the same free storage pool (and there is no fragmentation problem), Gafni's algorithm may still consume unbounded memory, because a process can only cancel items within itself. Consider a K-process simulation, where $K \geq 3$. Let $ts(e_i) = i$. Initially, an event $e_{(i-1)K+1}$ is scheduled for process p_i, $1 \leq i \leq K$. After $e_{(i-1)K+1}$ is executed, $K-1$ events $e_{(i-1)K+2}, e_{(i-1)K+3}, ..., e_{iK}$ are scheduled for p_i. The executions of these events do not create any new event. The space complexity of the sequential simulation is $O(K)$. We show that, in the worst case, the space complexity for Gafni's protocol is $O(KM_s) = O(K^2)$ even if the free storage is shared by all processes. Without loss of generality, we only consider the storage used in input queues. Suppose that every process is executed by a processor, and $(i+0.5)K$ units (where $1 \leq i < K$) of memory are available for the simulation. Consider the following scenario: At time t, processes $p_{K-i+1}, ..., p_K$ have completed execution, and iK units of memory are occupied by the events in their input queues. Process $p_j, 1 \leq j \leq K-i$, has not completed the execution of event $e_{(j-1)K+1}$. Thus, only $0.5K$ units of memory are left for $p_1, .., p_{K-i}$ to compete. Since the execution for e_1 has not been completed, fossil collection does not produce any free memory. Thus, processes $p_1, ..., p_{K-i}$ must cancel items within themselves. However, Gafni's protocol can at most produce

```
arrive()
    /* Let Γ be the set of items with send times larger than GVT */
1   if fossil collection does not reclaim any free storage then
2       atomic if Γ ≠ ∅ then
3           select an item e ∈ Γ and cancel it;
            /* The cancellation operation is the same as the case statement */
            /* in Gafni's algorithm */
4       else fail("memory exhaustion");
        end if
    end if
```

Figure 3: The *arrive* Procedure in the Cancelback Protocol.

$0.5K$ units of free memory, and the simulation fails. Hence, we have the following lemma.

Lemma 5 Time Warp simulation with Gafni's protocol may consume non-constant bounded memory.

Jefferson proposed an optimal protocol called *cancelback* in a shared memory architecture. In this protocol, if a process p needs storage for an item u, the protocol assumes that u is always allocated, but after the allocation, the system may not have enough free storage to continue the simulation. If so, a procedure *arrive* (cf. Figure 3) is invoked to generate more free storage. The main difference between this protocol and Gafni's protocol is that in cancelback, it is possible to cancel items in processes other than the processes that request free storage. In other words, cancelback is implemented in the system level with interrupt capability.

In this protocol (as well as Gafni's protocol), only items with send times larger than GVT can be cancelled. Since the history of a process with timestamps earlier than GVT may have been discarded by fossil collection, the items with send times earlier than GVT cannot be re-produced. The items with send times equal to GVT cannot be cancelled, otherwise cancelback may be trapped in an infinite loop [9]. Consider a two-process Time Warp simulation. Suppose that at time t, (i) $ck_1(t) = ck_2(t) = GVT(t)$, (ii) both processes request memory, and (iii) $F(t) = 0$. If all items in the system have send times no larger than $GVT(t)$, then these two processes will cancel an item with send time equal to $GVT(t)$ in each other (suppose that such items exist), re-execute, and cancel each other again. This action will repeat forever.

Although cancelback is considered as a complete solution for the storage management problem in Time Warp, the algorithm may not be implemented efficiently. In the following sections, we propose an optimal algorithm called *artificial rollback*. We show that this algorithm is easy to implement and analyze. We also give an implementation of artificial rollback, which is integrated with processor scheduling to adjust memory consumption rate based on the amount of free storage available in the system.

4 Artificial Rollback Protocol

We first introduce the concept of artificial rollback.

Definition 4 Without receiving a straggler, a process p may (on purpose) roll back its computation to a timestamp τ earlier than its local clock. Process p is said to *artificially* roll back to timestamp τ.

Lin and Lazowska [11] showed that artificial rollback does not affect the behavior of a Time Warp simulation (i.e., with or without artificial rollback, Time Warp produces the same result).

In several aspects, artificial rollback is equivalent to cancelback: The cancellation of an input message e is equivalent to an artificial rollback of the sender of e. The cancellation of a process state or an output message of a process is equivalent to an artificial rollback of that process. We introduce the concept of artificial rollback for the following reasons.

Artificial rollback is easy to implement. An artificial rollback is exactly the same as a normal rollback. Thus, the implementation of artificial rollback adds little overhead to the Time Warp mechanism. The protocol avoids several inefficiencies in the cancelback implementation. Examples are listed below.

- To determine which item to cancel, cancelback needs to access the inner structures (i.e., input queues, output queues, and state queues) of processes (cf. Line 2 of Figure 3). This may interfere or interrupt the executions of processes. On the other hand, using the concept of artificial rollback, an arbitrary timestamp larger than GVT can be selected to which to roll back an arbitrary process without accessing the inner structures of the process.

- To detect memory exhaustion, all items in the system must be examined by an atomic instruction (Line 2 of Figure 3). This operation is very expensive. We will show that in the artificial rollback protocol, memory exhaustion can be easily detected (cf. Lemma 11).
- Cancelback requires message preemption [15] while a protocol based on artificial rollback does not (cf. Section 5).
- Inefficiency may occur due to *busy cancelback*: Jefferson [8] showed that a reverse message may cause the sender to roll back, re-execute, and then resend, only to find that there is still not enough buffer space, and the sender must roll back, re-execute, and resend again. Another type of busy cancelback is due to the fact that the *arrive*() procedure is invoked independently by processes. Two processes (with local clocks later than GVT) may cancel an item of each other, re-execute, and cancel each other again. Although busy cancelback cannot repeat forever, it must be avoided for the purpose of improving performance. To solve this problem, we integrate artificial rollback with the processor scheduling policy (cf. Section 5).
- In cancelback, if an input message e in a process p_i is canceled, it is also sent in the reverse direction to cancel its antimessage e^- in another process p_j. If the simulation runs in a distributed environment with non-FIFO communication delay, then confusion may arise: Suppose that p_i receives e^- after e is sent back, then there are two possibilities. (i) Message e^- is the antimessage of e, and p_i should ignore it (because the positive copy is already cancelled). (ii) The antimessage of e is already annihilated, but p_j reproduces another negative message e^-, and this message is sent to p_i because of another rollback at p_j. In this case, p_i should not ignore the arriving e^-. Unfortunately, p_i cannot distinguish case (i) from case (ii) without extra treatment. In artificial rollback, there is no concept of canceling messages. Thus, this problem is avoided.

Artificial rollback is easy to analyze. Intuitively, if fossil collection is performed up to GVT, and all processes roll back to GVT, then the items left in the system are the same as those in the corresponding sequential simulation at simulation time GVT. Thus, it is easy to show that the space complexity of a protocol based on artificial rollback is $O(M_s)$ in a shared memory architecture.

Now we show how to design an optimal memory management protocol based on artificial rollback. We assume that the simulation is run on a shared memory architecture. Suppose that there are K processes and L processors in a Time Warp simulation. In a shared memory architecture, memory freed from a processor can be claimed by another processor. Let $\Psi_i(\tau, t) = \{e \in I_i(t), ts(e) = \tau\}$ and $\Psi(\tau, t) = \bigcup_{1 \le i \le K} \Psi_i(\tau, t)$. Let $\psi^+(e)$ be the set of events scheduled due to the execution of event e and $\psi^-(e)$ be the set of negative messages created due to the execution of event e. I.e., $\psi^-(e) = \{e^- | e^-$ is the antimessage of e^+ where $e^+ \in \psi^+(e)\}$. Thus, $|\psi^-(e)| = |\psi^+(e)|$.

Definition 5 A *system artificial rollback* or *system AR* with simulation time τ at time t is composed of two parts. In the first part, a fossil collection is performed up to $\tau \le GVT(t)$. In the second part, some processes are selected to be (artificially) rolled back. These processes are rolled back to timestamp no earlier than τ.

Lemma 6 Consider a Time Warp simulation. Suppose that a system AR is performed with τ at time t, and all processes are artificially rolled back to τ. If the system AR is completed at time t^+, then $M_{tw}(t^+)$ is bounded as

$$M_{tw}^-(\tau) \le M_{tw}(t^+) \le M_{tw}^+(\tau), \quad \text{where}$$

$$M_{tw}^-(\tau) = \sum_{1 \le i \le K} |x_i| + |E(\tau)| \quad \text{and} \tag{4}$$

$$M_{tw}^+(\tau) = \sum_{1 \le i \le K} |x_i| + |E(\tau)| + 2 \left| \bigcup_{e \in \Psi(t)} \psi^+(e) \right| \tag{5}$$

Proof:

From Corollary 1, if fossil collection is performed up to τ, then

$$\forall e \in I_i(t^+) \Rightarrow ts(e) \ge \tau, \quad \forall e \in O_i(t^+) \Rightarrow ts'(e) \ge \tau, \quad \forall x \in X_i(t^+) \Rightarrow ts(x) \ge \tau_i^- \tag{6}$$

If all processes artificially roll back to τ at time t, then all negative messages with send times later than τ are sent to annihilate the corresponding positive messages. In other words, after all rollbacks

complete, all messages have send times no later than τ, and all process states have timestamps no later than τ. Thus,

$$\forall e \in I_i(t^+) \Rightarrow ts'(e) \leq \tau, \quad \forall e \in O_i(t^+) \Rightarrow ts'(e) \leq \tau, \text{ and } \quad \forall x \in X_i(t^+) \Rightarrow ts(x) \leq \tau \tag{7}$$

Hence, (7) and (6) imply that

$$I_i(t^+) = \{e \in I_i(t) | ts'(e) \leq \tau, ts(e) \geq \tau\} \tag{8}$$

$$O_i(t^+) = \{e \in O_i(t) | ts'(e) \leq \tau, ts'(e) \geq \tau\} \tag{9}$$

$$X_i(t^+) = \{x \in x_i(t) | ts(x) \leq \tau, ts(x) \geq \tau_i^-\} = \{x(\tau)\} \text{ or } \{x(\tau_i^-)\} \tag{10}$$

Based on Theorem 1, $\Psi(\tau, t^+) = \Psi(\tau, t) = \{e \in \Psi | ts(e) = \tau\}$. After t^+, no new messages with timestamps τ will be created, and none of the events in $\Psi(\tau, t^+)$ will be cancelled. There are three cases:

Case I $GVT(t) > \tau$. From Definition 2, executions of all $e \in \Psi(\tau, t)$ have been completed by time t. Thus, (9) implies that

$$O_i(t^+) = \{e \in O_i(t) | ts'(e) = \tau\} = \bigcup_{e \in \Psi_i(\tau, t)} \psi^-(e) \tag{11}$$

Note that if $\Psi_i(\tau, t) = \emptyset$, then $\bigcup_{e \in \Psi_i(\tau,t)} \psi^-(e) = \emptyset$.

From Corollary 2 and (8), $I_i(t^+) = I_{i,\tau}(t) = \{e \in \Psi_i | ts'(e) \leq \tau, ts(e) \geq \tau\}$. Thus,

$$\begin{aligned} \bigcup_{1 \leq i \leq K} I_i(t^+) &= \{e \in \Psi | ts'(e) \leq \tau, ts(e) \geq \tau\} \\ &= \{e \in \Psi | ts'(e) < \tau, ts(e) \geq \tau\} \cup \{e \in \Psi | ts'(e) = \tau\} \\ &= E(\tau) \bigcup \left(\bigcup_{e \in \Psi(\tau, t)} \psi^+(e) \right) \end{aligned} \tag{12}$$

Note that $X_i(t^+) = \{x_i(\tau_i^-)\}$ or $\{x_i(\tau)\}$ but not both. From (10), (11) and (12), and because $|\psi^-(e)| = |\psi^+(e)|$, we have

$$M_{tw}(t^+) = \left| \bigcup_{1 \leq i \leq K} X_i(t^+) \right| + \left| \bigcup_{1 \leq i \leq K} I_i(t^+) \right| + \left| \bigcup_{1 \leq i \leq K} O_i(t^+) \right| = M_{tw}^+(\tau)$$

Case II $\tau = GVT(t)$ and none of events $e \in \Psi(\tau, t)$ have been executed at time t. We have

$$\forall e \in O_i(t) \Rightarrow ts'(e) < \tau \tag{13}$$

From (9) and (13),

$$O_i(t^+) = \{e \in O_i(t) | ts'(e) < \tau, ts'(e) \geq \tau\} = \emptyset \tag{14}$$

From (8) and Corollary 2, $I_i(t) = \{e \in \Psi_i | ts'(e) < \tau, ts(e) \geq \tau\}$, which implies

$$\bigcup_{1 \leq i \leq K} I_i(t) = \{e \in \Psi | ts'(e) < \tau, ts(e) \geq \tau\} = E(\tau) \tag{15}$$

From (10), (14), and (15), we have $M_{tw}(t^+) = M_{tw}^-(\tau)$.

Case III $\tau = GVT(t)$ and some (but not all) events in $\Psi(\tau, t)$ have been executed. The amount of memory consumed at t^+ is between that in Case I and Case II; that is, $M_{tw}^-(\tau) \leq M_{tw}(t^+) \leq M_{tw}^+(\tau)$.

From Cases I, II, and III, we have $M_{tw}^-(\tau) \leq M_{tw}(t^+) \leq M_{tw}^+(\tau)$. ∎

Note that even if $M_{tw}(t^+) < M_{tw}^+(\tau)$, the Time Warp simulation needs storage no less than $M_{tw}^+(\tau)$ to advance GVT after t^+. Thus, the minimal amount of storage required to complete a Time Warp simulation is

$$M_{tw,\min} = \max_{\forall \tau} M_{tw}^+(\tau) \tag{16}$$

Since $M_s = \max_{\forall \tau} M_s(\tau)$, from (5) and (16), and let C_1 and C_2 be two constants, we have

$$M_{tw,\min} = C_1 M_s + C_2 \tag{17}$$

In other words, if we perform artificial rollback frequently, then the space complexity of a Time Warp simulation is $O(M_s)$, the same as the corresponding sequential simulation. Note that C_1 and C_2 are small. If the size of a message in Time Warp is the same as the size of an event in sequential simulation, then $C_1 = 1$.

However, several issues must be considered in implementing the artificial rollback protocol:

Issue 1 When to perform artificial rollback.

Issue 2 Which process to roll back.

Issue 3 How far to roll back.

For Issue 1, a natural choice is to perform artificial rollback when the amount of free storage is not large enough to satisfy the request of a process at time t. Thus, in this paper we define a Time Warp simulation with artificial rollback as follows.

Definition 6 In a Time Warp simulation with artificial rollback (TWAR), a system AR (cf. Definition 5) is invoked with timestamp $GVT(t)$ at time t if and only if $F(t)$ is not large enough to satisfy the request of a process at t.

Let M be the amount of storage available for the simulation.

Constraint 1 If $M \geq M_{tw,\min}$ then TWAR eventually completes.

Constraint 2 If $M < M_{tw,\min}$ then TWAR terminates when the system exhausts storage.

If both Constraints 1 and 2 are satisfied, then from (17), a TWAR is optimal. With arbitrary decisions for Issues 2 and 3, TWAR may be trapped in an infinite loop (either Constraint 1 or Constraint 2 is not satisfied) or TWAR may terminate due to false "memory exhaustion" detection (i.e., Constraint 1 is not satisfied).

Definition 7 A TWAR implementation is *correct* if and only if both Constraints 1 and 2 are satisfied.

Now we propose an approach to implement a correct TWAR. Consider Constraint 1.

Lemma 7 If $M \geq M_{tw,\min}$ then Constraint 1 is satisfied if and only if when a process with local clock GVT requests storage, it is eventually satisfied.

Proof: TWAR eventually completes $\Leftrightarrow$ if a process requests storage, it is eventually satisfied $\Leftrightarrow$ if a process p_i with local clock GVT requests storage, it is eventually satisfied. ∎

Consider a TWAR. Suppose that a process p_i executes an event e at time t. Let $\Delta_i(t)$ be the total amount of storage required to complete the execution for e after (and including) t, and let $\delta_i(t)$ be the amount of storage requested by process p_i at time t. If p_i does not need any storage at t then $\delta_i(t) = 0$. If p_i requests storage, then it can only request the amount for one process state or one message at a time; i.e., $\delta_i(t) = |x_i(t)| = |x_i|$ or $|e|$ (we assume that the sizes of all messages in the

Time Warp simulation are the same). Thus, $\delta_i(t) \leq \Delta_i(t)$, and if the free storage is sufficient large, then there exists $t' \geq t$ such that $\delta_i(t') = \Delta_i(t')$ (i.e., if p_i's request is satisfied at time t', then after t', p_i will not require any storage to complete e's execution). If the execution of e is started at t, then $\Delta_i(t) = |x_i(t)| + |\psi^+(e)| + |\psi^-(e)| = |x_i(t)| + 2|\psi^+(e)|$. Let $\mathcal{P}(\tau, t)$ be the set of processes that execute events $e \in \Psi(\tau, t)$. Suppose that a system AR is performed with $\tau = GVT(t)$ at time t and all processes roll back to $GVT(t)$. If the system AR is completed at time t^+, then

$$M_{tw}(t^+) + \sum_{p_i \in \mathcal{P}(\tau,t)} \Delta_i(t) = M_{tw}^+(GVT(t)) \tag{18}$$

Let $P(t)$ be the set of processes that request memory when a system AR is performed at time t. Let $P_{GVT}(t) = \{p_i | p_i \in P(t), ck_i(t) = GVT(t)\}$. (That is, $P_{GVT}(t) = P(t) \cap \mathcal{P}(GVT(t), t)$.) For a process p_j with $ck_j(t) \neq GVT(t)$, its memory request is not critical to the progress of Time Warp. Thus, $\sum_{p_i \in P_{GVT}(t)} \delta_i(t) = \delta(t)$.

Definition 8 A system AR performed at time t is *effective* if and only if (i) the AR is with timestamp $GVT(t)$ (i.e., the fossil collection is performed up to $GVT(t)$) and (ii) after the AR is performed (i.e., at time t^+)

$$F(t^+) \geq \sum_{p_i \in P_{GVT}(t)} \delta_i(t) = \delta(t)$$

From Definition 8 and Lemma 7, we have the following lemma.

Lemma 8 A TWAR satisfies Constraint 1 if every system AR is effective and after the AR is performed, all processes $p_i \in P_{GVT}(t)$ obtain storage first.

Definition 9 Let $\alpha = \max\left[\max_{1 \leq i \leq K}(|x_i|), |e|\right]$. A system AR performed at time t is *strong* if (i) $F(t^+) \geq K\alpha$ or (ii) $F(t^+) < K\alpha$ and all processes are rolled back to $GVT(t)$.

Note that a process cannot request more storage than α units at a time.

Definition 10 A Time Warp simulation with strong artificial rollback (TWSAR) is a TWAR such that for all t, any system AR performed at t is strong, and all processes in $P_{GVT}(t)$ obtain storage first after the AR is performed.

Lemma 9 Consider a TWAR. If $M \geq M_{tw,\min}$ then a strong system AR is effective.

Proof: Suppose that a system AR is performed at time t. Since $\delta_i(t) \leq \alpha$ for all $p_i \in P_{GVT}(t)$, we have $\sum_{p_i \in P_{GVT(t)}} \delta_i(t) \leq K\alpha$. If $F(t^+) \geq K\alpha$ then $F(t^+) \geq \sum_{p_i \in P_{GVT}(t)} \delta_i(t)$ and the AR is effective. Suppose that $F(t^+) < K\alpha$, then all processes are rolled back to GVT(t) (Definition 9). From (18), we have

$$M_{tw}(t^+) + \sum_{p_i \in P_{GVT}(t)} \delta_i(t) \leq M_{tw}(t^+) + \sum_{p_i \in P_{GVT}(t)} \Delta_i(t) = M_{tw}^+(\tau) \leq M_{tw,\min}$$

Since $M_{tw}(t^+) = M - F(t^+)$, and $M \geq M_{tw,\min}$, we have $F(t^+) \geq \sum_{p_i \in P_{GVT}(t)} \delta_i(t)$ and the AR is effective. ∎

From Definition 10 and Lemmas 8 and 9, we have

Lemma 10 TWSAR satisfies Constraint 1.

Now we show that TWSAR satisfies Constraint 2.

Lemma 11 Consider a TWSAR. A strong system AR is not effective if and only if $M < M_{tw,\min}$.

Proof: Lemma 9 shows that, in TWSAR, if a system AR is not effective then $M < M_{tw,\min}$. We only need to prove that a system AR is not effective if $M < M_{tw,\min}$. Since $M < M_{tw,\min}$, there exists τ' such that

$$M_{tw}^-(\tau') \leq M < M_{tw}^+(\tau') \tag{19}$$

Let τ be the earliest timestamp that satisfies (19). Then $M \geq M_{tw}^+(\tau'')$ for all $\tau'' < \tau$. From Lemma 10, TWSAR progresses until $GVT(t) = \tau$ for some time t. Suppose that a strong system AR is performed at time t. Since $M < M_{tw}^+(\tau)$, we have $F(t^+) < \sum_{p_i \in P_{GVT}(t)} \Delta_i(t)$. Thus, at some time $t' \geq t$ after a finite number of strong system ARs have been performed, we have $F(t'^+) < \sum_{p_i \in P_{GVT}(t')} \delta_i(t')$, and the last AR is not effective. ∎

Since we can easily test whether a system AR is effective in TWSAR, memory exhaustion can be effectively detected. Thus, from Lemmas 10 and 11, we have

Theorem 2 Every TWSAR is correct.

Theorem 2 suggests one way to implement a Time Warp simulation with the space complexity of $O(M_s)$ in a shared memory architecture. However, TWSAR cannot guarantee constant bounded memory usage in a distributed environment. In such an environment, we assume that there are J sites. In site j, where $1 \leq j \leq J$, L_j processors execute K_j processes, where $\sum_{1 \leq j \leq J} L_j = L$ and $\sum_{1 \leq j \leq J} K_j = K$. Theoretically, we can find a modified AR protocol with space complexity $O(JM_s)$. However, to achieve this space complexity, intensive synchronizations among sites may be required, which may significantly increase the time complexity. Thus, instead of proposing an optimal memory management protocol, we describe a simple protocol similar to the TWSAR for the shared memory architecture: The strong system AR is performed within a site, and α is replaced by $\alpha_j = \max\left[\max_{p_i \in P_j}(|x_i|), |e|\right]$, where P_j is the set of processes executed in site j. If a strong system AR within a site is not effective, then the processor exhausts its memory and the simulation terminates.

5 An Implementation for the Artificial Rollback Protocol

This section gives one implementation for the artificial rollback protocol in a shared memory architecture. In this implementation, artificial rollback is integrated with a non-work-conserving processor scheduling algorithm to avoid the busy cancelback phenomenon in the cancelback protocol. In this policy, the number of active processors is dynamically adjusted according to the amount of free storage in the system. The variables used in the algorithm are listed in Table 1. A global variable is accessed *atomically* in an **atomic** operation. A global variable can be accessed simultaneously by two processes if each of them executes a non-atomic operation (usually a "read" to the variable), and no other process accesses the variable via an atomic operation. Constants L and K represent the numbers of processors and processes respectively, where $L \leq K$. Variable F represents the amount of free storage in the system. Every processor runs a schedule procedure (cf. Figure 4) independently until the simulation completes (i.e., the variable *eos=true* in Line S1; *eos* is set *true* when distributed termination condition is detected by some algorithm) or the system exhausts memory (i.e., $fail = true$; $fail$ is set *true* when variable $success = false$ in Line R11, Figure 5). If the simulation terminates due to memory exhaustion, then all processes are killed

α, β	**[constant]** α is the maximal amount of storage that a process may request at a time. $\beta = K\alpha$.
b_i	**[global]** $b_i = on$ iff process p_i requests storage when it enters the AR mode.
$ck^+(Q)$	**[local]** The largest local clock of processes in Q.
eos	**[global]** $eos = true$ iff the simulation completes.
F	**[global]** The amount of free storage in the system.
$fail$	**[global]** $fail = true$ iff the memory is exhausted.
$flag_{ar}$	**[global]** $flag_{ar} = true$ iff the system is in the AR mode.
GVT	**[global]** The computed global virtual time.
l	**[global]** The number of processors that have entered the AR mode.
n	**[local]** The amount of storage requested by a process. $n \leq \alpha$.
p_{id}, pr_{id}	**[local]** p_{id} is the id of a process, and pr_{id} is the id of a processor.
P_{GVT}	**[global]** $P_{GVT} = \{i \mid ck_i = GVT, b_i = on\}$.
Q_0	**[local]** $Q_0 = \{p_i \in Q_1 \mid ck_i \geq ck_j, \forall p_j \in Q_1\}$
Q_1	**[global]** The set of processes that have not been selected to (artificially) roll back.
Q_2	**[local]** The set of processes selected to roll back.
Q_3	**[global]** The set of processes in Q_2 that have not been rolled back to $ck^+(Q_1)$.
Q_e, Q_r	**[global]** p_i is in Q_e iff it is being executed by some processor. Otherwise, it is in Q_r.
$success$	**[local]** $success = true$ iff the memory request of a process is satisfied.

Table 1: The variables used in procedures *request*, *schedule* and *AR*.

(cf. Line S13). A process p_i belongs to one of the two sets Q_e and Q_r. Process p_i is in Q_e if it is being executed by some processor. Otherwise, it is in Q_r. The **select** operation (cf. Line S2) removes a process from Q_r. The process is executed by a processor (i.e., the process joins the set Q_e). The **execute** operation (cf. Line S9) executes a process according to the Time Warp protocol. The completion of the **execute** operation depends on the processor scheduling policy used. For example, in a round-robin policy, the process returns the control to the scheduler when the time quantum expires. A process may also give up the control when it invokes the procedure *block* (cf. Line R7). A processor is in one of four modes:

Normal rollback (NR) mode. (Lines S2-S3) The processor always executes a process that will perform roll back (if any); that is, the processes that receive stragglers have highest priority to execute. After a process rolls back, it returns control to the scheduler.

Artificial rollback (AR) mode. (Lines S5-S6) If the system cannot satisfy the memory request of a process (i.e., $flag_{ar} = true$ in Line S4; $flag_{ar}$ is set *true* if the system cannot allocate enough storage for a process; cf. Lines R4, R5, and R6 in Figure 5), then the procedure AR is invoked to (artificially) roll back some processes to reclaim more storage.

Aggressive execution (AE) mode. (Lines S8-S9) If the amount of free storage F is above a threshold β then all processors are active. In our design, $\beta = K\alpha$ where $\alpha = \max\left[\max_{1 \le i \le K}(|x_i|), |e|\right]$.

Conservative execution (CE) mode. If $F < \beta$, then the number of active processors is $\left\lceil \frac{F}{\alpha} \right\rceil$.

The priorities for the executions of the modes are NR > AR > AE (CE). When a process needs n units of memory, it executes the procedure *request* (cf. Figure 5). Normally, $P_{GVT} = \emptyset$ (we will discuss R0 later). Consider Line R1. If $F \ge n$ then the *allocate* procedure satisfies the request, and assigns *success* the value *true*. Otherwise, fossil collection is performed (Line R3). If $F < n$ after fossil collection, then some processes must be (artificially) rolled back to make more room. The process sets $flag_{ar} = true$ (Line R6) which forces the system to enter the AR mode. The process blocks itself at Line R7 (i.e., the process gives up the execution right and is queued on Q_r). In the AR mode, all processors stop executing processes. Instead, they cooperate to roll back processes (i.e., all processors execute Lines S5-S6 eventually). The system exits from the AR mode if

(i) $F > \beta$, or

```
schedule()
/* Initially eos = fail = false, l = 0 and flag_ar = false */
S1   while (not eos) and (not fail) do
S2        atomic select a process p_i with a straggler message;
S3        if p_i exists then roll back p_i; /* normal rollback mode */
S4        elseif flag_ar = true then /* artificial rollback mode */
S5             atomic l = l + 1;
S6             AR();
S7        elseif F ≥ β then /* aggressive execution mode */
S8             atomic select a process p_i;
S9             execute p_i;
S10       elseif atomic |Q_e| < ⌈F/α⌉ then  /* conservative execution mode */
S11            atomic select a process p_i;
S12            execute p_i;
          end if;
     end while
     if fail then  kill all processes;
```

Figure 4: The *schedule* procedure.

(ii) all processes except the ones with the smallest local clocks have been rolled back.

In other words, a strong system AR is completed when the system transfers from the AR mode to the AE/CE mode. Let us consider the AR mode in detail. When a processor enters the AR mode, the variable l increments by 1 (cf. Line S5 in Figure 4), and the procedure AR is invoked (cf. Line S6). Before a processor executes Line A3, it makes sure that all the other processors are in the AR mode (i.e., $l = L$ at Line A1, and all processes are in Q_r).

In the procedure AR, Q_1 is the set of processes that have not yet been selected to (artificially) roll back, and Q_2 is the set of processes selected to roll back. Thus, $Q_1 \cup Q_2 = Q_r$ and $Q_1 \cap Q_2 = \emptyset$. Initially, Q_2 is empty (before Procedure AR is executed), and $Q_1 \leftarrow Q_r$ (Line A3). The processor with processor id $pr\_id = 1$ is the coordinator that determines the set of processes to be rolled back. In Lines A4-A5, the set Q_0 of processes with the largest local clocks in Q_1 are moved to Q_2. Then all processes in Q_2 are rolled back to the timestamp $ck^+(Q_1)$, the largest local clock of any process in Q_1. This procedure continues until either conditions (i) or (ii) are satisfied. The set Q_3 consists of processes in Q_2 that have not been rolled back to $ck^+(Q_1)$. Before Line A7 is executed, Q_3 is empty. At Line A7, $Q_3 \leftarrow Q_2$. The *remove* function atomically removes a process p_i from

```
/* At the beginning of the simulation, P_GVT = ∅, fail = flag_ar = false, and b_i = off */
/* Variable success is an output parameter returned by request */
request(n)
R0  while P_GVT ≠ ∅ do nop;
R1  allocate(n, success);/* success is an out parameter */
R2  while not success do
R3        compute GVT and perform fossil collection;
R4        allocate(n, success);
R5        if not success then
R6              b_i = on; flag_ar = true;
R7              block();
R8              if P_GVT = ∅ then allocate(n, success);
R9              elseif p_id ∈ P_GVT then
R10                   allocate(n, success);
R11                   if not success then  fail = true;
R12                   else atomic P_GVT = P_GVT − {p_id};
R13             else
R14                   while P_GVT ≠ ∅ do nop;
R15                   allocate(n, success);
                end if;
R16             b_i = off;
          end if;
    end while
```

Figure 5: The procedure *request*.

```
AR()
     /* Initially, Q_2 = Q_3 = ∅, and P_GVT = ∅ */
A1   while l < L do nop;
A2   if pr_id = 1 then /* the coordinator */
A3        Q_1 = Q_r;
A4        Q_0 = {p_i ∈ Q_1 | ck_i ≥ ck_j, ∀p_j ∈ Q_1};
A5        move Q_0 from Q_1 to Q_2;
A6        while (F < β) and (|Q_1| > 1) do
A7             atomic Q_3 = Q_2;
A8             while atomic remove(p_i, Q_3) do roll back p_i to ck^+(Q_1);
               /* Q_3 = ∅ */
A9             Q_0 = {p_i ∈ Q_1 | ck_i ≥ ck_j, ∀p_j ∈ Q_1};
A10            move Q_0 from Q_1 to Q_2;
          end while;
A11       Q_2 = ∅;
A12       l = 0;
A13       P_GVT = {i | ck_i = GVT and b_i = on};
A14       flag_ar = false;
     else /* a process other than the coordinator */
A15       while flag_ar do
A16            while atomic remove(p_i, Q_3) do roll back p_i to ck^+(Q_1);
          end while;
     end if
```

Figure 6: The procedure AR.

Q_3. This function returns *true* if p_i exists, otherwise it returns $false$. The coordinator (Line A8) and the other processors (Line A16) roll back all processes in Q_2. When the coordinator executes Lines A9-A10 and A6-A7, Q_3 is empty, and the other processors keep testing the condition at Line A16 and do nothing. After the coordinator exits from the loop A6-A10, $Q_2 \leftarrow \emptyset$ and $l \leftarrow 0$ (Lines A11 and A12) and $flag_{ar} \leftarrow false$ (Line A14; we will elaborate more on Line A13 later). At Line A15, $flag_{ar} = false$, all processors leave the AR mode and enter the execution mode.

The bit b_i is on if p_i blocks itself when the system enters the AR mode (Line R6). The variable P_{GVT} represents the set of the processes that have local clocks GVT[6] when they execute Line R3. It is not difficult to see that these processes resume their executions at Line R8. The blocked processes re-execute the *allocate* procedure (Lines R8-R15). If $F < n$, then the while loop (Lines R2-R15) repeats, or the process announces memory exhaustion (Line R11), and terminates the simulation. In Lines R8-R15, processes in P_{GVT} must obtain memory first (cf. Definition 10). Thus a process not in P_{GVT} must wait until the processes in P_{GVT} are all served (Lines R0 and R14). (Note that a process that executes R0 is not in P_{GVT}). If a process in P_{GVT} cannot obtain the amount of storage it needs, then the simulation exhausts memory (cf. Lemma 11) and terminates. Note that if there is enough free storage, then the execution of *request* is very efficient. In such a case, $P_{GVT} = \emptyset$ in Line R0, the while loop is executed once, and the process exits from the *request* procedure after R1 is executed.

6 Conclusion

This paper studied the space complexity of parallel simulation. The goal is to design an efficient optimal memory management protocol which guarantees that the memory consumption of parallel simulation is of the same order as sequential simulation. We first derived the relationships among the space complexities of sequential simulation, Chandy-Misra simulation, and Time Warp simulation. We showed that Chandy-Misra may consume more storage than sequential simulation, or vice versa. Then we showed that Time Warp never consumes less memory than sequential simulation. Then we describe cancelback, a previously proposed optimal memory management protocol for Time Warp. Although cancelback is considered as a complete solution for the storage management problem in Time Warp, some efficiency issues in implementing this algorithm must be considered. In this

[6] GVT is the global virtual time computed at Line R3, not the current global virtual time.

paper, we proposed an optimal algorithm called *artificial rollback*. We showed that our algorithm is easy to implement and analyze. An implementation of artificial rollback was given, which is integrated with processor scheduling to adjust memory consumption rate based on the amount of free storage available in the system.

With the artificial rollback protocol, Time Warp is able to reduce the amount of memory used in parallel simulation (while other simulation approaches such as the Chandy-Misra protocol cannot). However, the progress of the simulation may degrade. The trade-off between time and space is still an open question. Several Studies [1, 25] have been conducted to investigate this issue.

Acknowledgement

We would like to thank Ian Akyildiz, Richard Fujimoto, Will Leland, David Jefferson, Victor Mak, Peter Reiher, and the four anonymous reviewers for their valuable comments.

References

[1] Akyildiz, I.F., Chen, L., Das, S.R., Fujimoto, R.M., Serfozo, R.F. Performance Analysis of Time Warp with Limited Memory. To appear in *1992 ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*.

[2] Chandy, K.M., and Misra, J. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.

[3] Fujimoto, R.M. Performance Measurements of Distributed Simulation Strategies. *Transactions of the Society for Computer Simulation*, 6(2):89–132, April 1989.

[4] Fujimoto, R.M. Time Warp on a Shared Memory Multiprocessor. *Proc. 1989 International Conference on Parallel Processing*, Volume III:242–249, August 1989.

[5] Fujimoto, R.M. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):31–53, October 1990.

[6] Gafni, A. Rollback Mechanisms for Optimistic Distributed Simulation. *Proc. 1988 SCS Multiconference on Distributed Simulation*, pages 61–67, February 1988.

[7] Jefferson, D. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[8] Jefferson, D. Virtual Time II: The Cancelback Protocol for Storage Management in Time Warp. *Proc. 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 75–90, August 1990.

[9] Jefferson, D. Private communication. 1991.

[10] Kaudel, F. J. A Literature Survey on Distributed Discrete Event Simulation. *Simuletter*, 18(2):11–21, June 1987.

[11] Lin, Y.-B., and Lazowska. Design Issues on Optimistic Distributed Simulation. Submitted for publication, 1991.

[12] Lin, Y.-B., and Lazowska, E.D. Determining the Global Virtual Time in a Distributed Simulation. *Proc. International Conference on Parallel Processing*, III:201–209, 1990.

[13] Lin, Y.-B., and Lazowska, E.D. Exploiting Lookahead in Parallel Simulation. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):457–469, October 1990.

[14] Lin, Y.-B., and Lazowska, E.D. Optimality Considerations for Time Warp Parallel Simulation. *Proc. 1990 SCS Multiconference on Distributed Simulation*, pages 29–34, January 1990.

[15] Lin, Y.-B., and Lazowska, E.D. A Study of Time Warp Rollback Mechanisms. *ACM Transactions on Modeling and Computer Simulation*, 1(1), 1991.

[16] Lin, Y.-B., and Lazowska, E.D. A Time-Division Algorithm for Parallel Simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(1), 1991.

[17] Lin, Y.-B., Lazowska, E.D., and Baer, J.-L. Conservative Parallel Simulation For Systems With No Lookahead. Technical Report 89-07-07, Department of Computer Science and Engineering, University of Washington, July 1989.

[18] Lin, Y.-B., Lazowska, E.D., and Bailey, M.L. Comparing Synchronization Protocols for Parallel Logic-Level Simulation. *Proc. International Conference on Parallel Processing*, III:223–227, 1990.

[19] Lipton, R.J., and Mizell, D.W. Time Warp vs. Chandy-Misra: A Worst-Case Comparison. *Proc. 1990 SCS Multiconference on Distributed Simulation*, pages 137–143, January 1990.

[20] Loucks, W.M., and Preiss, B.R. The Role of Knowledge in Distributed Simulation. *Proc. 1990 SCS Multiconference on Distributed Simulation*, pages 9–16, January 1990.

[21] Madisetti, V., Walrand, J., and Messerschmitt, D. Synchronization in Message-passing Computers – Models, Algorithms, and Analysis. *Proc. 1990 SCS Multiconference on Distributed Simulation*, pages 35–48, January 1990.

[22] Misra, J. Distributed Discrete-Event Simulation. *Computing Surveys*, 18(1):39–65, March 1986.

[23] Nicol, D.M. Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks. *Proc. ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 124–137, 1988.

[24] Preiss, B.R. Performance of Discrete Event Simulation on a Multiprocessor Using Optimistic and Conservative Synchronization. *Proc. International Conference on Parallel Processing*, III:218–222, 1990.

[25] Preiss, B.R., MacIntyre, I.D., and Loucks, W.M. On the Trade-off between Time and Space in Optimistic Parallel Discrete-Event Simulation. *Proc. 6th Workshop on Parallel and Distributed Simulation*, 1992.

[26] Reynolds, P.F. Heterogeneous Distributed Simulation. *Proc. 1988 Winter Simulation Conference*, pages 206–209, December 1988.

[27] Righter, R., and Walrand, J.C. Distributed Simulation of Discrete Event Systems. *Proceedings of the IEEE*, 77(1), January 1989.

[28] Samadi, B. *Distributed Simulation, Algorithms and Performance Analysis*. PhD thesis, Computer Science Department, University of California, Los Angeles, 1985.

[29] Wagner, D.B., and Lazowska, E.D. Parallel Simulation of Queueing Networks: Limitations and Potentials. *Proc. 1989 ACM SIGMETRICS and Performance '89 Conference*, pages 146–155, 1989.

A Notation

This section serves as an index to the notation used in this paper. Every notation is followed by either **s** (sequential), **cm** (Chandy-Misra), or **tw** (Time Warp) which indicates the protocol where the notation is defined.

$|\cdot|$ [**s**,**cm**,**tw**] $|\cdot|$ is the amount of memory required to store the item "$\cdot$".

α [**tw**] $\alpha = \max\left[\max_{1\le i\le K}(|x_i|), |e|\right]$.

$ck_i(t)$ [**tw**] $ck_i(t)$ is the local clock of process p_i at time t or the timestamp of the event executed by p_i at time t.

$\Delta_i(t)$ [**tw**] Suppose that an event e is executed by process p_i at time t. $\Delta_i(t)$ is the total amount of storage required by p_i to complete the execution for e after (and including) t.

$\delta_i(t)$ [**tw**] Suppose that an event e is executed by process p_i at time t. $\delta_i(t)$ is the amount of storage requested by process p_i at time t. If p_i does not need any storage at t then $\delta_i(t) = 0$. If p_i requests storage, then it can only request the amount for one process state or one message at a time; i.e., $\delta_i(t) = |x_i|$ or $|e|$.

$\delta(t)$ [**tw**] $\delta(t)$ is the minimal amount of storage required to make the Time Warp simulation progress at time t. $\delta(t) = \sum_{p_i \in P_{GVT}(t)} \delta_i(t)$.

e [**s**,**cm**,**tw**] e is an event.

$E(\tau)$ [**s**] $E(\tau) = \{e \in \Psi | ts'(e) < \tau, ts(e) \ge \tau\}$.

$F(t)$ [**tw**] $F(t)$ is the amount of free storage available for Time Warp simulation at time t.

Γ_p, Γ [**tw**] Γ_p is the set of items in process p with send times larger than GVT in Time Warp, and $\Gamma = \bigcup_{\forall p} \Gamma_p$.

$GVT(t)$ [**tw**] $GVT(t)$ is the global virtual time of Time Warp at time t.

$I_i(t)$ [**cm**,**tw**] $I_i(t)$ is the set of events scheduled for p_i at time t; i.e., the events already in p_i's input queue, and the events that have been sent from other processes, but not yet received by p_i.

$I_{i,\tau}(t)$ [**tw**] For $\tau_{FC}(t) \leq \tau \leq GVT(t)$, $I_{i,\tau}(t) = \{e \in \Psi_i | ts'(e) < \tau, ts(e) \geq \tau_{FC}(t)\}$.

K [**s**,**cm**,**tw**] K is the number of processes in the simulation.

L [**cm**,**tw**] L is the number of processors in the simulation.

$M_{tw}^{-}(\tau)$ [**tw**] $M_{tw}^{-}(\tau) = \sum_{1 \leq i \leq K} |x_i| + |E(\tau)|$.

$M_{tw}^{+}(\tau)$ [**tw**] $M_{tw}^{+}(\tau) = \sum_{1 \leq i \leq K} |x_i| + |E(\tau)| + 2 \left| \bigcup_{e \in \Psi(t)} \psi^{+}(e) \right|$.

$M_{tw,\min}$ [**tw**] $M_{tw,\min} = \max_{\forall \tau} M_{tw}^{+}(\tau)$.

M [**s**,**cm**,**tw**] M is the amount of storage available for the simulation.

$\mathcal{M}_s(\tau)$ [**s**] $\mathcal{M}_s = \left[\bigcup_{1 \leq i \leq K} x_i(\tau_i^{-}), E(\tau) \right]$ is the sequential snapshot at simulation time τ.

$\mathcal{M}_{cm}(t)$ [**cm**] $\mathcal{M}_{cm}(t) = \bigcup_{1 \leq i \leq K} [x_i(t), I_i(t), O_i(t)]$ is a snapshot of Chandy-Misra at t.

$\mathcal{M}_{tw}(t)$ [**tw**] $\mathcal{M}_{tw}(t) = \bigcup_{1 \leq i \leq K} [x_i(t), I_i(t), O_i(t)]$ is the snapshot of Time Warp at t.

$M_s(\tau), M_s$ [**s**] $M_s(\tau) = |\mathcal{M}_s(\tau)|$ and $M_s = \max_{\forall \tau} M_s(\tau)$.

M_{cm} [**cm**] $M_{cm} = \max_{\forall t} |\mathcal{M}_{cm}(t)|$.

$M_{tw}(t), M_{tw}$ [**tw**] $M_{tw}(t) = |\mathcal{M}_{tw}(t)|$, and $M_{tw} = \max_{\forall t} M_{tw}(t)$.

$O_i(t)$ [**cm**,**tw**] $O_i(t)$ is the output queue of process p_i at time t.

p_i [**s**,**cm**,**tw**] p_i is a process in simulation.

$P(t)$ [**tw**] $P(t)$ is the set of processes that request memory when a system AR is performed at time t.

$P_{GVT}(t)$ [**tw**] $P_{GVT}(t) = \{p_i | p_i \in P(t), ck_i(t) = GVT(t)\}$.

$\mathcal{P}(\tau, t)$ [**tw**] $\mathcal{P}(\tau, t)$ is the set of processes that execute events $e \in \Psi(\tau, t)$.

τ_i^- [**s**,**tw**] $\tau_i^- = ts(e)$ such that $e \in \Psi_i$, $\tau_i^- < \tau$, and for all $e' \in \Psi_i$, $ts(e') < \tau \Rightarrow ts(e') \leq \tau_i^-$.

$\psi^+(e)$ [**s**,**tw**] $\psi^+(e)$ is the set of events scheduled due to the execution of event e.

$\psi^-(e)$ [**tw**] $\psi^-(e)$ is the set of negative messages created due to the execution of event e. I.e., $\psi^-(e) = \{e^- | e^-$ is the antimessage of e^+ and $e^+ \in \psi^+(e)\}$. Note that $|\psi^-(e)| = |\psi^+(e)|$.

$\Psi_i(\tau, t), \Psi(\tau, t)$ [**tw**] $\Psi_i(\tau, t) = \{e \in I_i(t), ts(e) = \tau\}$. $\Psi(\tau, t) = \bigcup_{1 \leq i \leq < K} \Psi_i(\tau, t)$.

Ψ_i, Ψ [**s**] Ψ_i is the set of events executed by p_i in the sequential simulation. $\Psi = \bigcup_{1 \leq i \leq K} \Psi_i$.

$\tau_{FC}(t)$ [**tw**] At time t, fossil collection is performed up to simulation time $\tau_{FC}(t) \leq GVT(t)$.

t^+ [**tw**] In the artificial rollback protocol, if a fossil collection is performed up to simulation time τ, and all processes are artificially rolled back to τ at time t, then the fossil collection and artificial rollback are completed at time t^+.

$ts(e)$ [**s**,**cm**,**tw**] $ts(e)$ is the timestamp of event e.

$ts'(e)$ [**s**,**cm**,**tw**] $ts'(e)$ is the send time of e. If the scheduling of e is due to the execution of e_0, then $ts'(e) = ts(e_0)$.

$ts(x)$	[**tw**] The timestamp of the process state x. $ts(x) = ts(e)$ if the process state of p_i is x after it executes an event e.
$x_i(\tau)$	[**s**] $x_i(\tau)$ is the state of process p_i after it has executed all events with timestamps no later than τ in the sequential simulation.
$x_i(t)$	[**cm**,**tw**] $x_i(t)$ is the state of process p_i at time t.
$\|x_i\|$	[**s**,**tw**] $\|x_i\|$ is the amount of memory required to store a state of process p_i.
$X_i(t)$	[**tw**] $X_i(t)$ is the set of states in the state queue of process p_i at time t.